NPSCS-93-011

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California

Software Merge: Semantics of Combining Changes to Programs[1]

by Valdis Berzins

December 1993

Approved for public release; distribution is unlimited.

Prepared for:

Army Research Office
Research Triangle Park, NC

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAR ADMIRAL T. A. MERCER
Superintendent

HARRISON SHULL
Provost

This report was prepared for and funded by the Army research Office under grant number ARO-145-91.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release;
distribution is unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
NPSCS-93-011

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Computer Science Dept.
Naval Postgraduate School

6b. OFFICE SYMBOL *(if applicable)*
CS

7a. NAME OF MONITORING ORGANIZATION
Army Research Office

6c. ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943

7b. ADDRESS *(City, State, and ZIP Code)*

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
Naval Postgraduate School

8b. OFFICE SYMBOL *(if applicable)*
NPS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
Grant number ARO-145-91

8c. ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943

10. SOURCE OF FUNDING NUMBERS

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| | | | |

11. TITLE *(Include Security Classification)*
Software Merge: Semantics of combining Changes to Programs[1]

12. PERSONAL AUTHOR(S)
Valdis Berzins

13a. TYPE OF REPORT
Interim

13b. TIME COVERED
FROM ______ TO ______

14. DATE OF REPORT *(Year, Month, Day)*
1993 December

15. PAGE COUNT
29

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

| FIELD | GROUP | SUB-GROUP |
|---|---|---|
| | | |
| | | |

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
software change merging, semantics, domains, software maintenance.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

We present a language-independent semantic model of the process of combining changes to programs. This model extends the domains used in denotational semantics (complete partial orders) to Boolean algebras, and represents incompatible modifications as well as compatible extensions. The model is used to define the intended semantics of change merging operations on programs and to establish some general properties of software merging. We determine conditions under which changes to subprograms of a software system can be merged independently and illustrate cases where this is not possible.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Valdis Berzins

22b. TELEPHONE *(Include Area Code)*
(408) 656-2461

22c. OFFICE SYMBOL
CSBe

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

# Software Merge: Semantics of Combining Changes to Programs[1]

*Valdis Berzins*

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

## *ABSTRACT*

We present a language-independent semantic model of the process of combining changes to programs. This model extends the domains used in denotational semantics (complete partial orders) to Boolean algebras, and represents incompatible modifications as well as compatible extensions. The model is used to define the intended semantics of change merging operations on programs and to establish some general properties of software merging. We determine conditions under which changes to subprograms of a software system can be merged independently and illustrate cases where this is not possible.

**Keywords:** software change merging, semantics, domains, software maintenance.

## 1. Introduction

Practical software systems have many different versions, all of which are constantly changing in response to changes in user needs, the operating environment, and the discovery of faults. Such changes often have to be developed concurrently and then combined, or the "same" change has to be applied to several different versions of the system. We call the process of combining several changes *software change merging*. Tool support for this process is desirable, possibly in the form of automated assistant to a designer [22]. However, a production-quality tool should always produce either a correct result or an indication of failure together with diagnostic information. A clear and precise model of change merging is needed to build such tools and to demonstrate that they achieve the required degree of "correctness".

This paper provides such a model and determines some of its characteristics. Since our goal is to formulate a language-independent definition of the requirements for change merging, we focus on merging changes to the *meaning* of a program (rather than changes to the concrete representation of a program). This is only part of the problem: a practical tool must operate on concrete representations of programs and produce a concrete program whose meaning agrees with our model unless it explicitly reports failure. Methods for change merging that operate on concrete representations of programs are outside the scope of this paper. A concrete method for change merging that is based directly on a special case of the models presented here can be found in [2].

[1] This research was supported in part by the Army Research Office under grant number ARO-145-91.

We achieve language-independence via the constructions used in denotational semantics to define the meanings of programming languages. The domains used in traditional denotational semantics do not have enough structure to provide a general definition of change merging. The main contribution of this paper is a set of extended domain constructions that can support such a definition and a formalization of the notion of a *semantic conflict* between changes. Since exact merging is not computable [1], the set of inputs for which a merging tool reports a failure and the semantic merging model produces a conflict-free result is one criterion for comparing different merging tools. We also explore some semantic limitations on merging that are related to functional decomposition.

Section 2 reviews some relevant previous work. Section 3 presents the construction of semantic domains appropriate for program merging, first reviewing the basic properties of approximation lattices, then extending these lattices to Boolean algebras and showing the relation to Browerian algebras. The purpose of these structures is to extend the ordinary semantic domains to include improper values representing combinations of incompatible design decisions. This lets us model software merging as a total operation on these extended domains, and enables the change merging in the domain of program meanings to identify semantic conflicts in cases where changes cannot be consistently combined. Section 4 uses the algebraic structures developed in Section 3 to support a formal definition of an ideal operation for combining the semantics of software modifications, and determines some of the properties of this formal model. Section 5 presents some conclusions and directions for future work.

## 2. Previous Work

The problem of combining two versions of a functional program was formalized as constructing the least common extension of two partial functions in [1]. This is a simplified version of the problem considered in this paper, which includes incompatible modifications as well as compatible extensions. The intended semantics of merging compatible extensions was expressed using lattices and the approximation ordering $\sqsubseteq$ used in traditional approaches to denotational semantics of programming languages [23, 25].

These lattice structures were refined into larger Boolean algebras to model incompatible changes via a suitable difference operation in [2]. The idea behind the domain construction was sketched briefly, and was limited to primitive domains, cross products, and a rough approximation to function spaces that was limited to additive functions (see [25] p. 105). The current paper repairs the deficiencies in the Boolean function space construction, and introduces Boolean versions of an extended sum constructor, two of the three main power domain constructors, and recursively defined domains (solutions to reflexive domain equations). The current paper also formalizes the idea of semantic conflicts, and explores properties of the change merging model related to functional composition as well as the conditions under which the result of a set of changes is independent of the order in which they are applied. We also show that change merging does not preserve monotonicity (or computability).

We view changes as transformations from versions to versions as was done in [2], and prove a minimality property suggested there. Change transformations have been developed in a different way in [20].

This paper characterizes the intended semantics of change merging but does not address concrete methods for change merging. Specific methods for change merging are briefly surveyed here (see [5] for more information). The first semantically based methods for combining two versions of a functional program were given in [1]. A method for merging versions of PROLOG programs is [24]. An approach to combining both modifications and compatible extensions to while-programs based on data flow analysis and operations on program dependency graphs is described by Horowitz, Prins, and Reps [12], and this approach was subsequently improved [26]. Another method for solving this problem that produces fewer spurious conflict reports is described in [2]. A method for merging changes to prototypes with concurrent actions and hard real-time constraints is described in [7].

## 3. Semantic Domains for Software Merging

This section explores semantic domains that can support change merging operations. A difference operation is desirable for this purpose, and at least a pseudo-difference operation seems to be needed. Since the domains commonly used in denotational semantics do not in general provide such operations, we explore refinements to those domains.

Section 3.1 reviews some relevant properties of the domains used in denotational semantics. Section 3.2 presents a construction that extends a class of complete partial orders commonly used in denotational semantics to *atomic Boolean algebras* (which provide a difference operation). Section 3.3 explores the relation of this effort to Browerian algebras (which provide a pseudo-difference operation).

Every atomic Boolean algebra is a Boolean algebra as well as a lattice, and satisfies all of the usual laws of Boolean algebras. It is possible to understand most of the later parts of this paper in terms of the interpretations for the Boolean operations summarized in Fig. 1, without knowledge of the special properties of the atomic Boolean algebras constructed in Section 3.2, although some parts of the proofs depend on these properties.

### 3.1. Lattices and Approximation Orderings

The domains used in denotational semantics are all partially ordered sets with an approximation ordering $\sqsubseteq$. Typical domains are special kinds of function spaces whose elements represent meanings of programs. An approximation $f \sqsubseteq g$ means that $g$ is a compatible extension of $f$: $g$ agrees with $f$ whenever $f$ is defined, and $g$ may be defined in cases where $f$ is not. Our model of change merging uses lattices that are special cases of this kind of structure.

A lattice is a partially ordered set that contains least upper bounds and greatest lower bounds for all finite subsets. Lattices have a least upper bound operation $\sqcup$, a greatest lower bound operation $\sqcap$, a least element $\bot$, and a greatest element $\top$.

All of the features of a lattice are relevant to change merging. The least upper bound merges all of the information contained in two elements. The greatest lower bound extracts the information common to both elements. The least element $\bot$ denotes the absence of information, and is used together with $\sqcap$ to indicate that two elements do not contain any common information. The greatest element $\top$ represents an inconsistency: the result of merging (all possible) incompatible information.

Denotational semantics was originally formulated using special kinds of lattices. Later formulations were recast in terms of cpos (complete partial orders) that are subsets of the original lattices. This was done partially to avoid overconstrained elements such as $\top$, which do not have a natural interpretation for individual programs. In the context of change merging, overconstrained elements represent conflicts between semantically incompatible changes. Since independently developed changes can easily be semantically incompatible, overconstrained elements have natural interpretations in our context.

Since lattice operations are applied to functions independently at each point, overconstrained elements may be useful for localizing semantic incompatibilities and diagnosing their causes. A simple example illustrates the idea.

$$[\bot, 1, 2, 3] \sqcup [4, 1, \bot, 5] = [4, 1, 2, 3 \sqcup 5]$$

Here we use sequences of length four as idealized finite examples of semantic functions representing meanings of programs. The least upper bound merges the information in the two functions to produce a third, which has normal data values in the first three positions and an overconstrained value in the fourth. The presence of the overconstrained value indicates a semantic conflict, its location indicates the part of the input space that is affected (the fourth value in the index set for the sequence), and the overconstrained value itself indicates the nature of the conflict (the same output has been simultaneously constrained to have two different and incompatible values, 3 and 5). Thus we have a semantic model for an idealized error reporting facility.

It is of course much easier to merge the semantics of programs than it is to materialize the concrete programs corresponding to the merged semantics, and the problem of diagnosing and locating conflicts between changes is far from being solved in practice. However, it does help to have a clear idea of an idealized goal for error reporting. A partial change merging method that can in some cases derive a program representation with overconstrained program elements in the parts of the program that produce semantic conflicts is described in [2]. Representations of such overconstrained program elements are detectable by syntactic operations and could be used as a basis for generating concrete error messages.

### 3.2. Atomic Boolean Algebras

To model incompatible changes to the semantics of a program, we need a difference operation to describe the information that was removed from a software object. If we treat the partial functions computed by our programs as sets of pairs, then the set difference operation captures this idea for first-order functions. Together with $\subseteq$, $\cup$, $\cap$ and set complementation this structure forms a Boolean algebra, which is a simple and typical example of the extended semantic domains we will be using.

Boolean algebras provide natural generalizations that cover higher-order functions as well. The motivation for our choice of algebraic structures is discussed further in Section 3.3. This section reviews the basic properties of Boolean algebras and shows how to construct the Boolean algebras for higher order function spaces.

There are many equivalent definitions of Boolean algebras [10]. Every Boolean algebra is a complemented distributive lattice with respect to the partial ordering defined by the relations $x \sqsubseteq y <=> xy = x <=> x + y = y$. In addition to the lattice operations a Boolean algebra has a complement operation, which can be used to define a binary difference operator $x - y = x\bar{y}$ that obeys the algebraic properties of set difference. These structures are important for our goals of finding minimal compatible extensions and finding minimal change transformations.

We use notations for operations on Boolean algebras common in circuit design. Unfortunately, these notations are not the same as those used for lattice operations in the context of denotational semantics. The correspondence is shown in Fig. 1 [2].

We will be working with a special class of Boolean algebras, those that are atomic. An *atom* is an element that is distinct from the bottom element $\mathbf{0}$ and has no lower bounds other than itself and $\mathbf{0}$. A Boolean algebra is *atomic* if every element is the least upper bound of the set of atoms it dominates (lemma 1, [10] p. 70).

Every atomic Boolean algebra is isomorphic to the power set of its atoms, which becomes a Boolean algebra when $\sqcup$, $\sqcap$, and complement are interpreted as the union, intersection, and complement operations on sets (theorem 5, [10]). This isomorphism demonstrates that the difference operator of a atomic Boolean algebra really is the same as set difference. It also implies that every atomic Boolean algebra is complete and that its structure is determined by its cardinality. A Boolean algebra (or any lattice) is *complete* if and only if it has least upper bounds and greatest lower bounds for arbitrary subsets, not just the finite ones.

| Lattice | Boolean Algebra | Interpretation |
|---|---|---|
| $\top$ | $\mathbf{1}$ | Conflict |
| $\bot$ | $\mathbf{0}$ | Undefined |
| $x \sqsubseteq y$ | $x \sqsubseteq y$ | Compatible extension predicate |
| $x \sqcup y$ | $x + y$ | Compatible combination |
| $x \sqcap y$ | $xy$ | Common part |
|  | $\bar{x}$ | Complement |
|  | $x - y$ | Difference |

**Fig. 1 Correspondence between Lattice Notation and Boolean Notation**

The Boolean algebras used in digital circuit design are atomic and finite: elements can be represented as fixed-length vectors of bits, the atoms are all zero except at one bit position, and the cardinality of the value set is equal to a power of two. The Boolean algebras we use for semantic domains are mostly function spaces, and the cardinality of the value set is typically infinite.

An example of a Boolean algebra that is neither atomic nor complete is the set of all finite unions of half-open real intervals of the form $\{x \mid a \leq x < b\}$ where $0 \leq a \leq b \leq 1$ and $a$ and $b$ are rational numbers. The operations of the algebra are ordinary set-theoretic unions, intersections, and complements, the ordering is subset, the bottom element is the empty set (any interval with $a = b$), and the top element is the interval with $a = 0$ and $b = 1$. To see that this Boolean algebra does not have any atoms, note that there is an infinite descending chain below every element other than the empty set. To see that it is not complete, note that any infinite set of pairwise disjoint intervals does not have a least upper bound.

We will find that isomorphic Boolean algebras can be given quite different interpretations, and that some properties relevant to change merging, such as whether or not a given element represents a semantic conflict, can depend on the intended interpretation as well as on the structure of the algebra. Since the intended interpretation of a domain is determined by how the domain was constructed, we implicitly label each domain with the operation that was used to construct it and consider algebras constructed in different ways to be distinct. To keep our conventions simple we will follow this convention uniformly and explicitly mention all isomorphisms, although this introduces some distinctions that are not significant. For example, the domains $A \times (B \times C)$ and $(A \times B) \times C$ are considered to be different even though they are isomorphic and have essentially the same meaning.

Our domain construction will proceed as follows. We will take some domains to be given a priori, and we will label them as primitive. All other domains will be constructed from the primitive domains using a fixed set of domain constructors and possibly reflexive domain equations. The domain constructors are cross products, extended disjoint sums, and function spaces. Section 3.3 extends this by adding power domain constructors.

### 3.2.1. Primitive Domains

The domains representing ordinary data types will be treated as primitive. By an ordinary data type we mean a set whose values are either completely defined or completely undefined. Ordinary data types can include composite data structures as long as it is not possible for some subcomponents to be defined while others are not. An example of a data type that is not ordinary is a list type with lazy evaluation for the element extraction operation, since some components might be well defined while an attempt to extract other components might cause the program to go into an infinite loop.

In denotational semantics, ordinary data types are represented as flat lattices or flat cpos. The Boolean algebra representing an ordinary data type is the power set of the type with the usual set operations.

Each element of this Boolean algebra is a set of values of the data type that represents the least upper bound of those values. Proper values of the type are represented as singleton sets of the algebra, and these are the atoms of the Boolean algebra. The approximation relation $\sqsubseteq$ is interpreted as the subset relation, and the operations $x + y$, $xy$, and $x - y$ are interpreted as union, intersection, and set difference operations. The completely undefined element $\mathbf{0}$ is represented as the empty set, and the completely overconstrained element $\mathbf{1}$ is represented as the set of all values of the data type.

For the primitive domains, semantic conflicts are represented by sets containing more than one element. An element $x$ of a primitive domain is *conflict-free* if and only $x \sqsubseteq a$ for some atom $a$. Note that the undefined element $\mathbf{0}$ is conflict-free as well as the proper values of the type.

The relation between the lattice construction and the Boolean algebra construction is illustrated in Fig. 2 [2] for a discrete type representing the states of a traffic light. Our construction preserves the structure of flat lattices everywhere except for the top element $\top$, which is refined into a set of distinct improper elements in the Boolean algebra. The Boolean algebra can support a total and single-valued difference operator because the least upper bounds of distinct sets of proper data elements are all distinct. It is not possible to define such a difference operator for flat lattices with more than two elements because all of these upper bounds are identified with the single element $\top$ in the flat lattice.

### 3.2.2. Domain Construction: Cross Products

We use the same product space constructor used in denotational semantics, since the product of two atomic Boolean algebras is an atomic Boolean algebra. The atoms of the product space are the tuples that have an atom as one component and undefined elements for all of the other components.

An element of a product space contains a semantic conflict if one of its components does: a tuple $t \in D_1 \times D_2$ is *conflict-free* if and only if $t_1$ and $t_2$ are conflict-free.

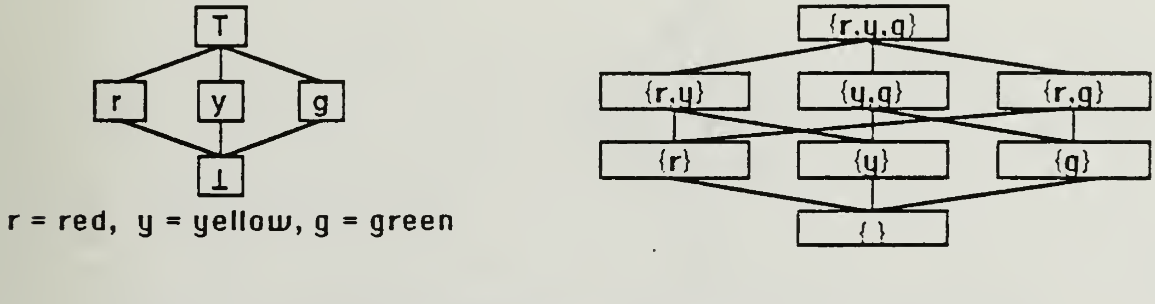


Fig. 2 A Flat Lattice and the Corresponding Boolean Algebra

Note that product spaces contain conflict-free elements that are not atoms and are not $0$. Since we can have a product space isomorphic to a primitive domain, we can see that the "conflict-free" property need not be preserved by isomorphism.

### 3.2.3. Domain Construction: Extended Sums

The domains produced by the classical disjoint sum construction are not in general Boolean algebras, nor can they be embedded in Boolean algebras. This is so because all Boolean algebras are distributive, but disjoint sums do not in general produce distributive lattices. To see this let $A$ and $B$ be atomic Boolean algebras and consider any elements $a, a' \in A$ and $b \in B$ that are ordered as shown in Fig. 3. Such elements exist whenever $A$ contains at least three distinct atoms and $B$ contains at least two distinct atoms. These elements fail to satisfy the distributive law:

$$a + (a' \; b) = a + 0 = a \neq a' = a' \; 1 = (a + a')(a + b).$$

Since an embedding must preserve least upper bounds and greatest lower bounds, these elements will also violate the distributive law in any other lattice in which the sum domain can be embedded.

We therefore reexamine disjoint sums in the context of change merging. To accurately represent semantic conflicts, we would like every pair of proper data values in our semantic domains to have a distinct least upper bound. This suggests that it is not really desirable to keep sum domains completely disjoint: certainly we want the parts of the domains containing the proper elements to be disjoint, but we would like to have distinct representations for all of the possible conflicts between the maximal proper elements from the different components of the sum. So we suggest using cross products under the usual ordering (Section 3.2.2) to encode extended disjoint sums.

This is not as unreasonable as it may at first appear: if we restrict our attention to the subset of the pairs that have an undefined element $0$ in at least one component, the ordering on the cross product space is exactly the same as the ordering on a coalesced

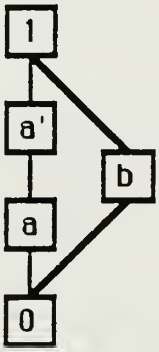


Fig. 3 Disjoint Sums Are Not Distributive

sum domain. The elements outside this subset all represent conflicts between elements from different components of the disjoint sum, and can be treated as not belonging to either component of the "sum". If we "encode" disjoint sums as product domains, the injection functions $\mathrm{in}_i$, the discrimination functions $\mathrm{is}_i$, and the projection functions $\mathrm{out}_i$ that are usually used in the definitions of semantic functions involving sum domains can be defined as follows.

$\mathrm{in}_1(x) = (x, 0)$ $\qquad$ $\mathrm{in}_2(x) = (0, x)$

$\mathrm{is}_1(p) = (p_1 \neq 0)\ \&\ (p_2 = 0)$ $\qquad$ $\mathrm{is}_2(p) = (p_1 = 0)\ \&\ (p_2 \neq 0)$

$\mathrm{out}_i(p) = \text{if } \mathrm{is}_i(p) \text{ then } p_i \text{ else if } p = (0, 0) \text{ then } 0 \text{ else } 1$

Thus the encoding appears to be a workable (though not very elegant) solution that allows domain constructions involving sum constructors to be simulated by using cross products, which do yield Boolean algebras.

An element of an extended sum domain contains a conflict if either component has a conflict or if we do not have a $0$ in at least one component: an element $x \in D_1 + D_2$ is *conflict-free* if and only if $x_1$ and $x_2$ are conflict-free and either $x_1 = 0$ or $x_2 = 0$.

### 3.2.4. Domain Construction: Function Spaces

The continuous function space constructor of denotational semantics does not preserve Boolean algebras, but the function space construction used in ordinary mathematics does. If $A$ and $B$ are atomic Boolean algebras, then the complete function space $A \rightarrow B$ is the atomic Boolean algebra whose elements are *all* of the functions from $A$ to $B$ with the natural pointwise ordering. The ordering, distinguished elements, and operations of the function space are described by the following relationships.

(a) $f \sqsubseteq g \iff \forall x \in A\, [f(x) \sqsubseteq g(x)]$

(b) $0(x) = 0$

(c) $1(x) = 1$

(d) $(f + g)(x) = f(x) + g(x)$

(e) $(f\ g)(x) = f(x)\ g(x)$

(f) $(\overline{f}\ )(x) = \overline{f(x)}$

(g) $(f - g)(x) = f(x) - g(x)$

The relation (a) is the standard ordering for function spaces in denotational semantics, and implies relations (b) - (g). The relations (b) and (c) describe the connection between the top and bottom elements of the function space $A \rightarrow B$ and the top and bottom elements of its range algebra $B$. The relations (d) - (g) are homomorphic extension rules which define the operations of the function space in terms of the operations of its range algebra. The operations are well defined because $B$ is complete, so that least upper bounds exist for all subsets of its atoms. The Boolean algebra properties are satisfied for the function space because they are satisfied by the values of the functions at each point in $A$.

The resulting function space is an atomic Boolean algebra. The atoms of $A \rightarrow B$ are functions $g[x, y]$ that are undefined at all points of the input space $A$ except for the point $x$, and have the value $y$ at that point, where $x$ can be any point in $A$ and $y$ can be any atom of $B$:

$g[x, y](z) = \text{if } x = z \text{ then } y \text{ else } 0.$

Every element of the function space is the least upper bound of the atoms it dominates because the corresponding property is true in $B$ for the values of those functions at every point in $A$, since $B$ is an atomic Boolean algebra.

The classical function space construction used in denotational semantics produces subsets of our function spaces. Note that we cannot limit the construction to just the monotonic functions (or just the continuous functions), because a Boolean algebra must be closed under complements, and the complement of a monotonic function is anti-monotonic.

Since our function spaces are complete, they satisfy the weaker completeness properties needed to show the existence of least fixed points of continuous functions. They also contain all of the continuous functions, and hence support recursive definitions for elements of the function spaces.

A function contains a conflict if its value at some meaningful point contains a conflict: a function $f: D_1 \to D_2$ is *conflict-free* if and only if $f(x)$ is conflict-free for all conflict-free $x \in D_1$. We restrict our attention to the conflict-free points in $D_1$ because computable (monotonic) functions can be expected to produce conflicts at overdefined points.

### 3.2.5. Domain Construction: Reflexive Domains

Reflexive domains are the solutions to recursive domain equations, in which equality is interpreted as lattice isomorphism. Since our function spaces contain all functions, our function space constructor strictly increases the cardinality of the space, so we cannot hope to find spaces that are isomorphic to their own function spaces. This is one of the classical problems in denotational semantics, and it implies that the function space construction of Section 3.2.4 cannot be used in recursive domain equations. However, if we are willing to restrict ourselves to distributive lattices and embed solutions to domain equations in larger Boolean algebras, there is a way out via the space of continuous functions used in denotational semantics.

We start with a lemma that relates properties of domain constructors to the properties of reflexive domains defined using those constructors.

**Lemma 1.** If $\mathbf{F}$ is a continuous domain constructor [23] that preserves distributiveness then the minimal solution to the domain equation $D = \mathbf{F}(D)$ exists and is distributive. Indeed, any property that is preserved by the domain constructors and can be expressed as an equation or inclusion between expressions denoting elements of a domain is satisfied in D.

**Proof:** $D$ exists by (theorem 4.2.8, [23]). The minimal domain $\perp$ contains only one point, so all inclusions and equations on elements are trivially satisfied, and in particular we have $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ and $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ for all $x, y, z \in \perp$. So the domain $\perp$ is distributive. Since $\mathbf{F}$ preserves distributiveness, the approximating domains $\mathbf{F}^n(\perp)$ are distributive for all natural numbers $n$, by induction.

By (lemma 4.2.5, [23]), $x \sqsubseteq y \iff \forall n\,[\pi_n^*(x) \sqsubseteq \pi_n^*(y)]$, where $\pi_n^*$ are the projection functions from $D$ into $\mathbf{F}^n(\perp)$ defined in [23]. (Two elements of the limiting domain are ordered if and only if their projections into all of the approximation domains are ordered in the same way). This implies $x \sqcup y = z$ in $D \iff \forall n\,[\pi_n^*(x) \sqcup \pi_n^*(y) = \pi_n^*(z)]$ and similarly for $\sqcap$. Since $x = y \iff x \sqsubseteq y \;\&\; y \sqsubseteq x$ we also have $x = y$ in $D \iff \forall n\,[\pi_n^*(x) = \pi_n^*(y)]$. By structural induction it follows that any equation among finite expressions built using variables, $\sqcup$, and $\sqcap$ is satisfied in the limit domain $D$ if and only if the expression with each variable $v$ replaced by the corresponding projection $\pi_n^*(v)$ it is satisfied in all the approximating domains $\mathbf{F}^n(\perp)$. So D must be distributive, since all of the approximating domains are distributive.

Now we can proceed to the main theorem.

**Theorem 1.** The solution to any domain equation composed of given Boolean algebras and domain constructors that preserve distributiveness can be embedded in a Boolean algebra.

**Proof:** Boolean algebras are distributive lattices by definition. Lemma 1 says that the solution to a domain equation composed of domain constructors that preserve distributiveness is also a distributive lattice. Every distributive lattice can be extended to a Boolean algebra ([16], p. 450 ff.). So the result follows.

The primitive domains of Section 3.2.1 are all Boolean algebras. Cross products and function spaces of distributive lattices are distributive (theorem 13, [17], Ch. 14). The lattice of continuous functions is distributive because it is a sublattice of the full function space. Extended disjoint sums are encoded as cross products in Section 3.2.3. So the solution to any domain equation expressed using these constructs will be a distributive lattice, which can be embedded in a Boolean algebra. This Boolean algebra is a good candidate for hosting change merging operations because it is a natural completion of the semantic domain that contains distinct representations for all possible conflict elements.

The solution to a reflexive domain equation is labeled with the outermost domain constructor on its right hand side, and the appropriate definition of conflict-free elements is determined accordingly.

These constructions allow us to formulate change merging operations for most of the programming languages covered by classical denotational semantics. This does not include the class of languages with parallel or nondeterministic operations, which are considered briefly in the next section.

### 3.3. Browerian Algebras

Boolean algebras are sufficient for modeling the semantics of change merging for the programming languages whose semantics can be expressed using the constructions of Section 3.2. Some work on concrete methods for change merging has been based on a more general class of Browerian algebras [21]. This section discusses the role of Browerian algebras in modeling the semantics of change merging.

A Browerian algebra is a lattice with a pseudo-difference operation $\dot{-}$ that satisfies the property $x \dot{-} y \sqsubseteq z <=> x \sqsubseteq y + z$. Every Boolean algebra becomes a Browerian algebra when it is equipped with a difference operator as defined in Section 3.2, but there exist Browerian algebras that are not Boolean algebras. In particular, Browerian algebras need not satisfy the law $\bar{\bar{x}} = x$. The difference operator of a Boolean algebra must satisfy both of the following properties.

P1. $(x - y) + (xy) = x$

P2. $(x - y)y = 0$

The pseudo-difference operator of a Browerian algebras must satisfy P1, which says that all of the information in $x$ is either contained in $y$ or is contained in the part of $x$ that is not in $y$: $(x \dot{-} y) + (xy) = ((x \dot{-} y) + x)((x \dot{-} y) + y) = x(x + y) = x$ if we note that $x \dot{-} y \sqsubseteq x$ and use (proposition A.12, [21]). However, the pseudo-difference operator may fail to satisfy P2, which says that the result of removing $y$ from $x$ does not contain any of the information in $y$. It is easy to see that this disjointness property holds in a Boolean algebra: $(x - y)y = x\bar{y}y = x0 = 0$. The pseudo-difference operator is thus not really a difference operator, because it may not remove all of the information contained in $y$, although it must remove as much as possible without violating property P1. This means that the underlying lattice of a Browerian algebra need not have sufficiently fine resolution to separate $x$ and $y$, in the sense that it may not be possible to remove all of the information in $y$ from $x$ without also removing some information in $x$ that is not contained in $y$.

We have worked with Boolean algebras because they provide a natural representation for precisely identifying the sources of semantic conflicts, proofs are simpler, and the extra generality of Browerian algebras is not needed to express the semantics of change merging for most programming languages. Separating the elements of the semantic domains in the sense of the previous paragraph is usually not an issue because the data values of most programming languages are disjoint (can be modeled as a flat cpo) or can be assembled from disjoint parts (using cross products or functions).

Browerian algebras were used in [21] because the concrete dependence graph representations of programs do not have a Boolean algebra structure. However, we note that the semantic correctness of the HPR algorithm [12] as well as of the variant in [21] is not derived from the Browerian algebra structure of the dependence graph representations of the programs, but rather from the concrete properties of program slices.

A realistic example of a semantic domain that should properly be modeled as a Browerian algebra is the domain of maximum execution times (METs) for a language with hard real-time constraints such as PSDL [8]. Since a larger number represents a weaker timing constraint, the approximation ordering $\sqsubseteq$ for this domain is the total ordering $\geq$ on numerical values. This is an extreme example of lack of separability because two constraints cannot be disjoint unless one of them is the vacuous constraint $\infty$ (the bottom element of the lattice). The lattice of MET values is a Browerian algebra with $x \dot{-} y = \text{if } x \sqsubseteq y \text{ then } \infty \text{ else } x$. Since every Browerian algebra is a distributive lattice (theorem 1.3, [19]), every Browerian algebra, including the MET domain, can be extended to a Boolean algebra by ([16], p. 450 ff.). A fragment of the MET domain and its embedding in a Boolean algebra are shown in Fig. 4, where proper elements have bold lines and the artificial elements added by the embedding have thin lines. Unlike the extended data domains of Section 3.2, where the artificial elements represent conflicts between distinct proper elements, the extra elements in this extension to a Boolean algebra provide artificial differences between proper elements that do not have realistic interpretations. Change merging in this Boolean algebra can result in an improper element, while change merging in the Browerian algebra yields a proper element that dominates the result in the Boolean algebra. For example, merging the change from 2 to 1 with the change from 2 to 3 according to the model explained in Section 4 yields the improper element $(1-2)+(1)(3)+(3-2)=3+(1-2)$ in the extended Boolean algebra and the dominating proper element $(1 \dot{-} 2)+(1)(3)+(3 \dot{-} 2)=1$ in the Browerian algebra, where $+$ and $-$ denote the operations from these algebras instead of the usual operations from arithmetic. This suggests Browerian algebras may provide appropriate change merging models for data domains with overlapping proper elements that cannot be separated into disjoint proper components.

A standard construction for Browerian algebras involves topological closure operations [18]. A topological closure operation on a Boolean algebra of sets is a function from sets to sets that satisfies the following properties for all sets $x$ and $y$ in the domain of the Boolean algebra:

(C1) $x \subseteq C(x)$,
(C2) $C(C(x)) = C(x)$,
(C3) $C(x \cup y) = C(x) \cup C(y)$ and
(C4) $C(\{\}) = \{\}$

Note that many "closure" operations commonly used in computer science are not topological closure operations. For example, the transitive closure of a graph does not satisfy condition C3. A set $x$ is closed relative to $C$ if $C(x) = x$. The closed elements

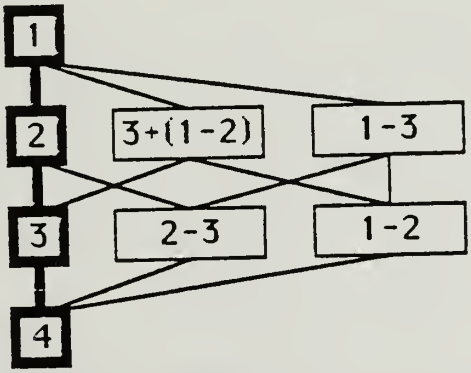


**Fig. 4 Extending the MET Browerian Algebra to a Boolean Algebra**

of any Boolean algebra of sets with a topological closure operation $C$ forms a Browerian algebra under the operation $x \dot{-} y = C(x - y)$ (theorem 1.14, [19]) and every Browerian algebra is isomorphic to an algebra constructed in this way (theorem 1.15, [19]). The connection to topology is that every topological space is a Browerian algebra and every Browerian algebra is isomorphic to a subalgebra of the algebra of closed sets of a topological space (theorem 1.19, [19]).

In particular, the set of all downwards closed subsets of any lattice forms a Browerian algebra under the construction outlined above. The downwards closure is defined by $DC(s) = \{y \mid \exists x \in s.y \sqsubseteq x\}$, and can easily be seen to satisfy the properties of a topological closure operation. Similar results hold for the upwards closure $UC(s) = \{y \mid \exists x \in s.x \sqsubseteq y\}$. These properties turn out to be relevant to power domains.

### 3.3.1. Domain Construction: Power Domains

We saw in Section 3.2 how the semantic domains for many classical programming languages can be naturally embedded in Boolean algebras. The constructions of Section 3.2 do not cover languages with parallelism and nondeterministic constructs, whose semantic domains usually include power sets (power domains). Several varieties of power domain constructions have been proposed for defining the semantics of programming languages with parallelism. The approximation orderings on these domains are only quasi-orderings, and the relation $x \equiv y <=> x \sqsubseteq y \; \& \; y \sqsubseteq x$ is an equivalence relation that is weaker than equality. The approximation relation becomes a partial ordering on equivalence classes in the quotient lattices with respect to this equivalence relation, but it is less cumbersome to represent each equivalence class by its largest member (the set with the most elements). There are three main variations on the power domain construction: the Hoare power domain, the Smyth power domain, and the Egli-Milner power domain. It turns out that the equivalence classes for the Hoare power domain are represented by the downwards closed subsets of the underlying lattice, those for the Smyth power domain are represented by the upwards closed subsets, and those for the Egli-Milner power domain are represented by the convex closed subsets [23].

Since the downwards closure and the upwards closure satisfy the topological closure properties C1-C4 above, the Hoare and Smyth power domains have a natural Browerian algebra structure, which implies that they are distributive lattices and can be extended to Boolean algebras. This implies that our model of the semantics of change merging can be applied to both the Hoare and Smyth power domains, and that theorem 1 can be applied to reflexive domain equations containing these power domain constructors. We leave open the question of whether the Browerian or the Boolean model of change merging is more appropriate for these domains.

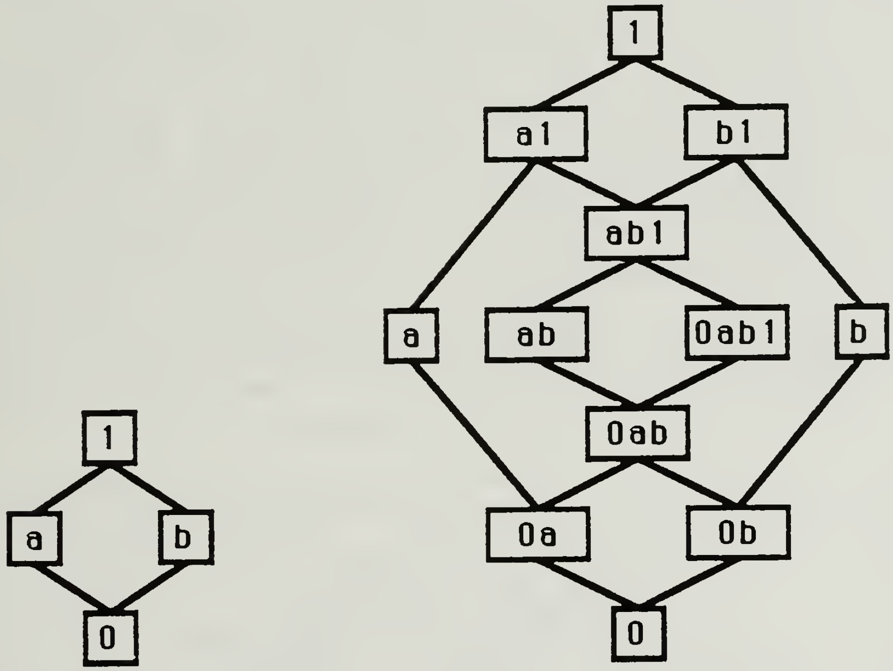
An element of a power domain represents a set of possible outcomes. Such an element has a semantic conflict if any of the possible outcomes have a conflict: a set $s \in P(D)$ is *conflict-free* if and only if $x$ is conflict-free for all $x \in s$.

We note that the convex closure $CC(s) = \{y \mid \exists x, z \in s.x \sqsubseteq y \sqsubseteq z\}$ does not satisfy the topological closure property C3 above, so it is not a topological closure operation and does not support the Browerian algebra construction. In fact, the Egli-Milner power domain is not in general a distributive lattice, as demonstrated by the following counterexample. Fig. 5. shows a four element Boolean algebra (a) and the Egli-Milner power domain derived from it (b). This power domain is not a distributive lattice because $\{a, b\} \sqcup (\{a, b, 1\} \sqcap \{a\}) = \{a, b\} \sqcup \{0, a\} = \{a, b\} \neq \{a, b, 1\} = \{a, b, 1\} \sqcap \{a, 1\} = (\{a, b\} \sqcup \{a, b, 1\}) \sqcap (\{a, b\} \sqcup (\{a\})$ where the least upper bounds and greatest lower bounds are taken with respect to the Egli-Milner ordering [23] shown in Fig. 5. (b). Since every Browerian algebra is a distributive lattice, Egli-Milner power domains cannot be extended to Browerian (or Boolean) algebras.

We conclude that none of the known models of change merging apply to languages whose semantics involve Egli-Milner power domains, since all of these models depend on (at least) a Browerian algebra structure. This issue is relevant to applications where programs that can fail to terminate must interact in a nondeterministic way (typically via parallel processing). The formulation of an appropriate model of change merging for this class of languages is left as an open problem.

## 4. Language-Independent Model of Software Merging

This section discusses some properties of the change merging operator in [2], which is defined as follows.



(a) Boolean Algebra (b) Derived Power Domain

Fig. 5 The Egli-Milner Power Domain is not Distributive

$$f\,[\,g]\,h = (\,f\ -\ g) + f\ \ h + (\,h -\ g) \qquad \text{(M1)}$$

We know that $(\,f\ -\ g) + f\ \ h + (\,h -\ g) = (\,f\ -\ g) + f\ \ g\,h + (\,h -\ g)$ because $f\ \ h = f\ \ h(\,g + \overline{g}\,) = f\ \ g\,h + f\,\overline{g}\ \ h$ and $f\,\overline{g}\ \ h \equiv \overline{g}\ \ h = (\,h -\ g)$ (this equivalence holds also for Browerian algebras, but the proof is longer). Thus our definition of $f\,[\,g]\,h$ coincides with the negmajority operation $f\,[\,g]\,h$ defined in [11] and the integration operation $f\,[\,g]\,h$ defined in [21]. The name "negmajority" was motivated by a propositional calculus interpretation of a Boolean operator that is true if and only if a majority of the three values is true after the middle value is negated. Our operator defines the same Boolean function, but it is interpreted in a much larger Boolean algebra representing the space of meaning functions for programs, rather than in the two-valued Boolean algebra of propositional calculus. Reps' version of $f\,[\,g]\,h$ is defined in terms of the pseudo-difference operation of a Browerian algebra, instead of the difference operation of a Boolean algebra, because the graphs used to represent programs in his algorithm do not satisfy all the properties of a Boolean algebra. However, we have seen in Section 3 that the semantic domains for sequential programs can be extended to Boolean algebras.

### 4.1. The Relation to Minimal Change Transformations

Informally, an operation for combining changes to functions should be able to apply the change defined by the difference between two versions $g$ and $f$ of a function to some other version $h$. In [2] we showed how to infer a mapping from versions to versions from the effect of a change on a particular base version $g$. This was done by characterizing the change from a function $g$ to a function $f$ in terms of the common part $f\ \ g$, the part that was added $f\ -\ g$, and the part that was removed $g - f$, as illustrated in Fig. 6. [2]. These components were shown to be disjoint and to contain all of the information in the functions $f$ and $g$, so that any change to a function can be characterized by the part that was added and the part that was removed.

We formalize the idea of a transformation $f\,[\,g]$ that changes a base version $g$ into a modified version $f$ as follows. We propose that change transformations should be functions of the form $\mathrm{T}[a,\,b]$ where $\mathrm{T}[a,\,b](x) = (x\ - a\,) + b$ for all $x$, and seek the smallest transformation such that $\mathrm{T}[a,\,b](\,g) = f$ (relative to the ordering defined by

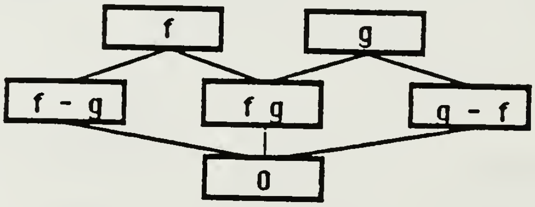


**Fig. 6 Characterizing Software Changes**

$T[a_1, b_1] \sqsubseteq T[a_2, b_2] \iff a_1 \sqsubseteq a_2 \;\&\; b_1 \sqsubseteq b_2$). This is based on the following assumptions:

(1) The information added as well as the information removed by a given change transformation should not depend on what version the change transformation is applied to, and

(2) The only additions and removals included in the change transformation should be those necessary to transform $g$ into $f$.

As was suggested in [2], these assumptions uniquely determine $f[g]$: $f[g] = T[g - f, f - g]$. To see that this change transformation maps $g$ into $f$ note that $T[g - f, f - g](g) = (g - (g - f)) + (f - g) = g\overline{g\overline{f}} + f\overline{g} = g\overline{g} + g\overline{\overline{f}} + \overline{g}f = 0 + (g + \overline{g})f = f$. To see that it is the smallest such transformation, suppose that $(g - a) + b = f$. Then $f - g = f\overline{g} = (g\overline{a} + b)\overline{g} = g\overline{g}\overline{a} + b\overline{g} = b\overline{g} \sqsubseteq b$ and $g - f = g\overline{f} = g\overline{(g\overline{a} + b)} = g(\overline{g} + a)\overline{b} = a\overline{b} \sqsubseteq a$. So $T[g - f, f - g] \sqsubseteq T[a, b]$.

This minimality property suggests that change merging should be defined by $f[g]\,h = (h - (g - f)) + (f - g)$. This was shown to be equivalent to (M1) in [2].

We now assess the adequacy of our formalization of the change merging process. By identifying programs with the functions they compute, we have ignored non-functional attributes such as computational efficiency and program understandability. The result of a change merging operation must certainly compute the correct function, and it may be subject to other constraints. These other constraints are beyond the scope of the current paper. Change merging that involves efficiency-improving modifications is addressed in [2], which suggests a method for program change merging that can accommodate efficiency improving changes as well as changes in program behavior. This method prefers program realizations from the changed versions over those in the base program in cases where several different realizations of the same semantic (sub-)function are present. The difficulty of recognizing efficiency-improving changes increases sharply with the size of the change and is undecidable in general. Automated capabilities in this area are necessarily limited, and can be augmented with declarations of programmer intent in the form of optional specifications for program fragments.

A related concern is the treatment of programs that do not terminate. Our formalization of change merging is intended to apply to any programming language that can be modeled by meaning functions over a fairly wide class of semantic domains, as described in Section 3.2. The effect of the change merging function defined in this section depends on the semantic model of the programming language being discussed. These models can differ considerably. In a simple imperative language whose programs denote functions from states to states, a program that immediately enters an infinite loop is equivalent to one that causes some state changes and then enters an infinite loop, because in both cases there is no final state, and hence there are no observable results. Thus a change from the first program to the second would have no effect on program behavior, since we would have $g - f = 0 = f - g$. For languages with I/O facilities, programs can denote functions from pairs (containing an input sequence and a state) to pairs (containing an output sequence and a state). In such a model, a program that enters an infinite loop immediately computes a different function than a program that first produces some output and then enters an infinite loop. A change from one to the other would have definite effect on program behavior in such a case. A different approach to specifying the semantics of programs that are not intended to terminate can be found in [3].

### 4.2. Change Merging does not Preserve Monotonicity

In Section 3 we motivated the need for function spaces containing functions that are not monotonic via a rather indirect argument. Now we are in a position to show that change merging can produce non-monotonic functions from monotonic ones. The following example shows three simple monotonic functions that produce a non-monotonic function via change merging.

| $x$ | $f(x)$ | $g(x)$ | $h(x)$ | $(f\,[g]\,h)(x)$ |
|---|---|---|---|---|
| 2 | 2 | 2 | $\bot$ | $\bot$ |
| $\bot$ | 2 | $\bot$ | $\bot$ | 2 |

Since every computable function is monotonic, this shows that the result of merging changes from computable functions to computable functions need not be computable. Thus the presence of conflicts is not the only factor that can lead to a situation where there does not exist any program that realizes the merge defined by a base program and two modified programs. This also shows that the domains of traditional denotational semantics must be extended to treat change merging as a total operation.

### 4.3. Commutativity and Order Dependence

Some of the known properties [11,21] of the change merging operation are shown below.

1. $b[a]c = c[a]b$
2. $(b[a]c)[a]d = b[a](c[a]d)$

If we treat $[a]$ as a binary operation, the first property says $[a]$ is commutative, and the second says that $[a]$ is associative. These properties imply that the order in which a set of changes to *the same base version* is combined does not matter.

We can also see that in general two change transformations commute whenever neither one removes information that the other adds. We can view the change merging operation as the application of a function $\mathrm{T}[a, b]$ to a program. Since $(f - g)(g - f) = 0$ we know that every change merging operation corresponds to a change transformation $\mathrm{T}[a, b]$ for which $ab = 0$. This says that in a minimal change transformation, none of the information that is removed is added back. We have

$$\mathrm{T}[a_2, b_2](\mathrm{T}[a_1, b_1](x)) = (((x - a_1) + b_1) - a_2) + b_2 = x\,\overline{a_1}\,\overline{a_2} + b_1\,\overline{a_2} + b_2,$$
$$\mathrm{T}[a_1, b_1](\mathrm{T}[a_2, b_2](x)) = (((x - a_2) + b_2) - a_1) + b_1 = x\,\overline{a_2}\,\overline{a_1} + b_2\,\overline{a_1} + b_1.$$

Clearly these are the same if $b_1\,\overline{a_2} = b_1$ and $b_2\,\overline{a_1} = b_2$. These conditions are equivalent to $b_1 \sqsubseteq \overline{a_2}$ and $b_2 \sqsubseteq \overline{a_1}$, which are equivalent to $b_1 a_2 = 0$ and $b_2 a_1 = 0$. So these conditions are sufficient for the transformations to commute. To see they are necessary, assume the transformations commute, and substitute $a_1$ for $x$:

$$a_1 \overline{a_1}\ \overline{a_2} + b_1 \overline{a_2} + b_2 = a_1 \overline{a_2}\ \overline{a_1} + b_2 \overline{a_1} + b_1,$$
$$0 + b_1 \overline{a_2} + b_2 = 0 + b_2 \overline{a_1} + b_1,$$
$$b_1 \overline{a_2} + b_2 = b_2 \overline{a_1} + b_1.$$

If we multiply both sides by $a_1$ we get

$$a_1 b_1 \overline{a_2} + a_1 b_2 = b_2 a_1 \overline{a_1} + a_1 b_1,$$
$$0 + a_1 b_2 = 0 + 0,$$
$$a_1 b_2 = 0.$$

Similarly, $a_2 b_1 = 0$. Thus we have shown that $f[g](t[u]x) = t[u](f[g]x)$ if and only if $(f - g)(u - t) = 0$ and $(t - u)(g - f) = 0$.

This result tells us what we must check to see whether a set of changes will give us the same result independently of the order in which the changes are merged. In case some changes override each other, in the sense that one removes information that another adds, then applying the same changes in a different order can intuitively be expected to give different results: the last transformation to be applied gets the final say about what should be done about the disputed information (to add or to remove). In such cases, human guidance is needed to resolve the relative priority between the two changes, possibly by going back to the requirements and the underlying justifications, and determining which change is more important.

## 4.4. Properties Related to Functional Composition

Independent modules are a basic requirement for programming on a large scale. We would like to have localized methods for analyzing and processing programs, so that the analysis can be decomposed into smaller independent subproblems. Such decoupling makes human understanding of large systems possible, as well as concurrent team efforts. It also leads to computational efficiency and opportunities for parallel computation in automated processing of programs. Thus to shed some light on the feasibility of change merging for large systems, we consider the interaction between program merging and functional composition, which we denote by $f \circ g$ where $(f \circ g)(x) = f(g(x))$. The urge to divide and conquer leads us to investigate the following distributive property.

(?) $(f_2 \circ f_1)[(g_2 \circ g_1)](h_2 \circ h_1) = (f_2[g_2]h_2) \circ (f_1[g_1]h_1)$

This expression describes two changes to an implementation that consists of two modules, where the output of the front end module $g_1$ is connected with the input of the back end module $g_2$. The left hand side represents the result of merging two changes to the system in one big operation, while the right hand side merges the changes to the front end and the changes to the back end in two independent operations and then connects the merged versions together. Although this distributive property appears to be plausible, it turns out to be false, as illustrated by the following example over the data values {1, 2, 3, 4}.

**Example 1.**

| $x$ | $f_2$ | $g_2$ | $h_2$ | $f_2[g_2]h_2$ | $f_1$ | $g_1$ | $h_1$ | $f_1[g_1]h_1$ | $f_2[g_2]h_2 \circ f_1[g_1]h_1$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 3 | 2 | 2 | 3 | 3 | 3 | 2 | 2 | 3 |
| 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 2 |
| 4 | 2 | 3 | 3 | 2 | 2 | 2 | 3 | 3 | 4 |

| $x$ | $f_2 \circ f_1$ | $g_2 \circ g_1$ | $h_2 \circ h_1$ | $(f_2 \circ f_1)[g_2 \circ g_1](h_2 \circ h_1)$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 4 | 4 | 2 | 2 |
| 3 | 2 | 3 | 3 | 3 |
| 4 | 3 | 2 | 4 | $3 \sqcup 4$ |

This example shows that the desired distributive property can be false even for well behaved functions and changes that do not conflict (compare the results for $x = 2$ or $x = 3$). This negative result is somewhat surprising, because it implies that changes to sub-functions can interfere even for purely functional programs. The interference effect demonstrated above is thus completely unrelated to classical concerns about coupling between modules due to side effects or state changes.

Although the interference effect comes from different coincidences of equal values in different versions of the functions, it is not caused by several different values being mapped to the same value by the individual functions: note that all of individual functions in Example 1 as well as their compositions and the results of the merges are one-to-one. Thus the problem lies in the relationships between the versions, and is unlikely to be preventable via localized restrictions that can be applied to each of the versions independently.

The circumstances where this interference effect shows up are complex and not very intuitive. For example, one case where the interference effect shows up without producing any local or global conflicts is when there is some input value $x$ for the front end for which the first change does not affect either the output of the front end ($f_1(x) = g_1(x)$) or the output of the entire system ($f_2(f_1(x)) = g_2(g_1(x))$), and the second change transforms the output of the front end into a value affected by the first change to the back end but not affected by the second change to the back end ($f_2(h_1(x)) \neq g_2(h_1(x)) = h_2(h_1(x))$).

A concrete example of this is a simple text formatter composed of two parts. The front end determines which words go on each line, and the base version puts as many words as will fit. The back end adds extra space between the words, and the base version adds enough space to make the right margin even on every line except the last line of a paragraph. The first change affects only the back end, which is changed to adjust the right margin of the last line of a paragraph in those cases where the space left at the end of the line is strictly less than a given tolerance T, where T > 1. The second change affects only the front end, which is changed so that if the space left on a line when no more words from the following text will fit is greater than the space left on the previous line plus the length of the last word on the previous line, then the last word of the previous line is moved up to the current line. The interference effect will show up for an input where the space left on the last line of the paragraph is exactly equal to the tolerance T, the previous line is completely full, and the last word on the previous line is one character long. If we merge the two changes in a single global operation then the merged version will move up the last word of the previous line and will not adjust the right margin of the last line. If we merge the two changes in isolation and combine the results, then the resulting program will adjust the right margin of the last line in addition to moving up the last word of the previous line. The two results are clearly different.

Also note that it is possible for the true merge of the compositions to contain a conflict even if the independent merges of the changes to corresponding subfunctions are conflict-free (compare the results of example 1 for $x = 4$). Approximate change merging methods must be safe to be practically useful: they must never silently turn a real conflict into a proper (but incorrect) result, although they may occasionally report potential conflicts that are not really there. This implies that unrestricted changes to different subprograms cannot in practice be merged independently, and that interprocedural analysis is necessary for reliable change merging. Thus correct divide and conquer methods for unrestricted change merging cannot exist, at least relative to modularizations based on functional decomposition.

This is quite unwelcome news for those concerned with the evolution of large software systems, because functional decomposition is the modularization principle underlying many widely used approaches to systems analysis and software design, such as structured analysis. We discuss the implications of this for the practitioner at the end of this section, after examining in more detail the conditions under which change merging can and cannot be performed independently for the components of functional decompositions.

Some weaker distributive properties related to change merging do hold. The following calculation shows that functional composition distributes over the program merge operation from the right.

**Property P3**

$$
\begin{aligned}
(f\,[\,g]\,h) \circ c &= (f\,\overline{g} + f\;h + \overline{g}\;h) \circ c \\
&= (f \circ c)(\overline{g} \circ c) + (f \circ c)(\,h \circ c) + (\overline{g} \circ c)(\,h \circ c) \\
&= (f \circ c)[\,g \circ c\,](\,h \circ c)
\end{aligned}
$$

The derivation uses the fact that $\overline{g} \circ c = \overline{g \circ c}$. This property says that the result of merging two changes is the same, independently of whether the inputs to the module that has been changed are supplied directly or come from another module. This is what most programmers would expect. We note that composition on the right may mask some conflicts in $f\,[\,g]\,h$, because the range of $c$ may be a strict subset of the domains of $f$, $g$, and $h$ that may not include some potential conflicts between those functions.

Functional composition does not always distribute over program merging from the left, but half of a distributive law is possible for conflict-free monotonic functions on primitive domains restricted to conflict-free portions of the input space. The restriction to monotonic functions is reasonable because every computable function is monotonic.

The restriction to conflict-free functions is also reasonable because conflicts can be introduced only by a process of combining changes that operates in an extended syntactic domain: for every programming language we have seen, every program that follows the syntax rules of the programming language is conflict-free. The restriction to conflict-free inputs is reasonable because there is no way to create self-contradictory input values on real computer systems. The restriction to primitive domains is less desirable, but it is necessary, as we demonstrate below. We start with a definition summarizing characterizations of semantic conflicts from Section 3 and a lemma.

**Definition**. An element $x$ of a primitive domain (Section 3.2.1) is *conflict-free* if and only if $x \sqsubseteq a$ for some atom $a$. A tuple $x \in D_1 \times D_2$ is *conflict-free* if and only if $x_1$ and $x_2$ are conflict-free. An element $x \in D_1 + D_2$ of an extended sum domain (encoded as a cross product) is *conflict-free* if and only if $x_1$ and $x_2$ are conflict-free and either $x_1 = 0$ or $x_2 = 0$. A function $f : D_1 \rightarrow D_2$ is *conflict-free* if and only if $f(x)$ is conflict-free for all conflict-free $x \in D_1$. A set $s \in P(D)$ is *conflict-free* if and only if $x$ is conflict-free for all $x \in s$, where $P(D)$ can denote either the Hoare power domain construction or the Smyth power domain construction (Section 3.3.1).

**Lemma 2.** If $f, g : D_1 \rightarrow D_2$ are restricted to the conflict-free portion of $D_1$, $c : D_2 \rightarrow D_3$ is monotonic, $f$ is conflict-free and $D_2$ is a primitive domain then $(c \circ f)(\overline{c} \circ g) \sqsubseteq (c \circ f)(c \circ \overline{g})$.

**Proof:**

Since $f$ is conflict-free, $f(x)$ is conflict-free for all conflict-free $x \in D_1$.
Let $x \in D_1$. Then $f(x) \sqsubseteq a$ for some atom a, since $D_2$ is a primitive domain.
Case 1: $a \sqsubseteq g(x)$. Then $f(x) \sqsubseteq g(x)$.
So $c(f(x)) \sqsubseteq c(g(x))$ and $c(f(x))\overline{c}(g(x)) = 0 \sqsubseteq c(f(x))c(\overline{g}(x))$.
Case 2: $a\ g(x) = 0$. Then $f(x)\ g(x) = 0$ since $f(x)\ g(x) \sqsubseteq a\ g(x)$.
Then $f(x) \sqsubseteq \overline{g}(x)$, $c(f(x)) \sqsubseteq c(\overline{g}(x))$, and $c(f(x)) = c(f(x))c(\overline{g}(x))$,
so $c(f(x))\overline{c}(g(x)) \sqsubseteq c(f(x)) = c(f(x))c(\overline{g}(x))$.
Since $a$ is an atom, cases 1 and 2 cover all possibilities.
So $(c \circ f)(\overline{c} \circ g) \sqsubseteq (c \circ f)(c \circ \overline{g})$ since $c(f(x))\overline{c}(g(x)) \sqsubseteq c(f(x))c(\overline{g}(x))$ for all $x$.

**Theorem 2.** If $f, g, h : D_1 \rightarrow D_2$ are restricted to the conflict-free portion of $D_1$, $c : D_2 \rightarrow D_3$ is monotonic, $f$ and $h$ are conflict-free, and $D_2$ is a primitive domain then $(c \circ f)[c \circ g](c \circ h) \sqsubseteq c \circ (f[g]h)$

**Proof:**

$$\begin{aligned}
&(c \circ f)[c \circ g](c \circ h) && \\
&\quad = (c \circ f)\overline{(c \circ g)} + (c \circ f)(c \circ h) + \overline{(c \circ g)}(c \circ h) && \text{Definition of . [ . ] .} \\
&\quad \sqsubseteq (c \circ f)(c \circ \overline{g}) + (c \circ f)(c \circ h) + (c \circ \overline{g})(c \circ h) && \text{Lemma 2} \\
&\quad \sqsubseteq c \circ (f \overline{g}) + c \circ (f\ h) + c \circ (\overline{g}\ h) && \text{Monotonicity of } c \\
&\quad \sqsubseteq c \circ (f \overline{g} + f\ h + \overline{g}\ h) && \text{Monotonicity of } c \\
&\quad = c \circ (f[g]h) && \text{Definition of . [ . ] .}
\end{aligned}$$

The following example shows that the inequality in theorem 2 is sometimes strict.

**Example 2.**

| f | g | h | c(1) | c(2) | c(3) |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 4 | 5 |

$$c(f)[c(g)]c(h) = 4[4]5 = 5$$
$$c(f[g]h) = c(1[2]3) = c(1 \sqcup 3) = c(1) \sqcup c(3) = 4 \sqcup 5 \neq 5$$

Together with property P3, theorem 2 says that if two changes affect *only one module* of a system and we merge the two changes to the module in isolation, then the result of replacing the original module with the doubly-changed one in the system is consistent with the (ideal) result of merging the corresponding versions of the entire system. The fact that theorem 2 is an inclusion rather than an equality means that some conflicts that appear when merging the changes to the module in isolation might not really be conflicts if the change merging operation were expanded to act on the context $c$ (the system in which the module is embedded) in addition to the base version $g$ and the two modified versions $f$ and $h$ of the impacted module. This seems natural enough, since the context $c$ is free to map two incompatible intermediate values $f(x)$ and $h(x)$ into the same value.

Theorem 2 behaves like a partial correctness result even though the inclusion appears to go in the wrong direction, because it is not possible in practice (under the hypotheses of the theorem) for the left hand side to be undefined and the right hand side to be a proper value unless the isolated merge $f[g]h$ produces a conflict. We can see this as follows. A strict inequality in theorem 2 is possible at a given point in $D_1$ only if the values of $f$, $g$, and $h$ are all distinct, because $c(f[g]g) = c(f) = c(f)[c(g)]c(g)$ and $c(f[g]f) = c(f) = c(f)[c(g)]c(f)$. If $c$ is computable we must have $c(\bot) = \bot$ except when $c$ is a constant function, and in that case theorem 2 becomes an equality. This gives us $c(\bot[g]h) = c(h) = \bot[c(g)]c(h) = c(\bot)[c(g)]c(h)$. Thus strict inequality is possible only when the values of $f$, $g$, and $h$ are all distinct and $f \neq \bot \neq h$, and in that case the value of $f[g]h$ at the given point represents a semantic conflict, since $D_2$ is a primitive domain.

Most programmers would expect these properties to be true in general. However, this is not the case: we have proved theorem 2 only when the intermediate domain $D_2$ is primitive, and it turns out that the theorem does not hold if we remove that restriction. To see that theorem 2 need not hold if we relax the requirement that $D_2$ is a primitive data domain, consider the following example, in which $D_1$ and $D_3$ are primitive domains containing numbers and $D_2$ contains 6-tuples of numbers.

**Example 3.**

$$T_1 = f(2) = [2, 2, \bot, \bot, \bot, 2] \qquad c(T_1) = 2$$
$$T_2 = g(2) = [2, \bot, 2, \bot, 2, \bot] \qquad c(T_2) = 2$$
$$T_3 = h(2) = [\bot, 2, 2, 2, \bot, \bot] \qquad c(T_3) = 2$$
$$T_4 = f(2)[g(2)]h(2) = [2, 2, 2, \bot, \bot, \bot] \qquad c(T_4) = 3 \neq 2 = c(T_1)[c(T_2)]c(T_3)$$

In this example the tuples $T_1$ - $T_4$ are all incomparable, so that monotonicity does not impose any relationships between their images under the function $c$. The fact that theorem 2 need not hold for all conflict-free monotonic functions may be even more surprising than Example 1, because it shows that there can be interference between functional composition and a *single* program change that produces a conflict-free result when carried out in isolation. This can happen if the intermediate domain $D_2$ contains partial functions or partial data structures (which are realized via lazy evaluation in some functional programming languages). Although the computations in Example 3 are somewhat unusual, the function $c$ can be realized (via several applications of a non-deterministic operation that evaluates two expressions in parallel and returns the value of the first one that terminates).

Some positive results are also possible for the simpler subproblem of combining compatible extensions to a program [1]. The simplification that occurs when merging compatible extensions is described by lemma 3.

**Lemma 3.** If $g \sqsubseteq f$ and $g \sqsubseteq h$ then $f[g]h = f + h$.

**Proof:**

$$\begin{aligned} f[g]h &= f\overline{g} + f\,g\,h + \overline{g}\,h \\ &= f\overline{g} + f\,g\,h + f\,g\,h + \overline{g}\,h \\ &= f\overline{g} + f\,g + g\,h + \overline{g}\,h && \text{since } g = f\,g = g\,h = f\,g\,h \\ &= f(\overline{g} + g) + (g + \overline{g})\,h \\ &= f + h \end{aligned}$$

The following distributivity properties hold for the special case of compatible extensions.

**Property P4**

$(f \circ c) + (g \circ c) = (f + h) \circ c$

**Property P5**

$(c \circ f) + (c \circ h) \sqsubseteq c \circ (f + h)$

Property P4 follows from the definition of + on function spaces. Property P5 follows from the monotonicity of $c$ (all computable functions are monotonic), and becomes an equality for additive (see [25] p. 105) functions $c$. The special case of additive functions has some practical interest: functions computed by conflict-free subprograms on primitive data domains are additive. The restriction to primitive domains (which also appears in theorem 2) holds for subsets of programming languages that do not allow subprograms to be passed as parameters, do not support partially defined data structures, and pass all parameters by value or by reference (i.e. no higher order functions, lazy evaluation, or call by name).

We can get half of a distributivity law for compatible extensions from properties P4 and P5.

**Theorem 3.** If $g_2 \sqsubseteq f_2, h_2$ and $g_1 \sqsubseteq f_1, h_1$ then $(f_2 \circ f_1)[g_2 \circ g_1](h_2 \circ h_1) \sqsubseteq (f_2[g_2]h_2) \circ (f_1[g_1]h_1)$.

**Proof:**

Suppose $g_2 \sqsubseteq f_2, h_2$ and $g_1 \sqsubseteq f_1, h_1$.
Then $g_2(g_1(x)) \sqsubseteq g_2(f_1(x)) \sqsubseteq f_2(f_1(x))$ for all $x$, so $g_2 \circ g_1 \sqsubseteq f_2 \circ f_1$.
Similarly $g_2 \circ g_1 \sqsubseteq h_2 \circ h_1$.

$$
\begin{aligned}
f_2 \circ f_1[g_2 \circ g_1]\, h_2 \circ h_1 &= (f_2 \circ f_1) + (h_2 \circ h_1) && \text{Lemma 3}\\
&\sqsubseteq (f_2 \circ f_1) + (f_2 \circ h_1) + (h_2 \circ f_1) + (h_2 \circ h_1) &&\\
&\sqsubseteq f_2 \circ (f_1 + h_1) + h_2 \circ (f_1 + h_1) && \text{Property P5}\\
&= (f_2 + h_2) \circ (f_1 + h_1) && \text{Property P4}\\
&= (f_2[g_2]\, h_2) \circ (f_1[g_1]\, h_1) && \text{Lemma 3}
\end{aligned}
$$

Theorem 3 can serve as the basis for an approximate method for merging compatible extensions that impact different modules of a system that are connected by data flows. This covers a situation in which methods based on program slicing, such as [12], would always report a conflict. Since theorem 3 is an inclusion rather than an equation, such a method may report some extraneous conflicts (see Example 2 and the following discussion).

Our results impact current software development practices in some environments. Common sense suggests that a change to a module can affect all of the other modules that can receive data from the modified module. However, in informal conversations, several practicing software engineers have described real projects where different people were routinely assigned to make concurrent changes to modules with apparently unrelated functions, and the results of such updates were combined simply by linking the new versions of both modules into the system after both updates appear to have been implemented correctly. This practice appears to be based on belief in the (incorrect) distributivity property marked with a (?) at the beginning of this section, or on the unquestioned assumption that if both engineers did their job correctly, then all should be well. The results of this section show that this assumption is not always sound, and that the situation is far from simple in the general case.

Example 1 shows that the results of combining changes to several different modules that are correct for each module in isolation can produce incorrect results when the modules are assembled into a system in such a way that the output of one changed module can reach the input of another changed module. More surprisingly, two changes to the same module that are consistent and produce correct results when the module is considered in isolation can produce an incorrect system-level result when the module is embedded in a larger system where nothing else has been changed (example 3), although the conditions under which this can occur are rather exotic.

Some special cases where independent changes can be combined in isolation and then can be safely embedded in a larger system are identified in theorems 2 and 3. In other cases, when changes are combined, all modules whose inputs can be affected by the outputs of the changed modules must be checked together to establish the mutual correctness of the combined changes. The correctness of the combined changes must be checked explicitly, in addition to checking the correctness of each change in isolation, since in the general case the combination can be incorrect even if each individual change produces correct results when considered in isolation form the other changes. The required check requires interprocedural analysis that can span large subsets of the system even if only a few modules of the system have actually been changed. Such checking can be quite expensive if it is to be done manually. This consideration provides some support for test strategies that use program slicing to determine the impact of a change and to guide the selection of test data.

It is natural to ask why this problem has not been noticed before, if it really is potentially serious. One relevant point is that the desired distributivity property is almost true, in the sense that it was fairly difficult to find counterexamples. We succeeded in doing so only by explicitly analyzing the merging formula and several failed attempts to prove that the property was true. Thus failures due to this mechanism might not be easy to detect using small sets of test cases. Another point is that large systems in practice are subject to failures that do not have clearly understood causes: "integration problems" during system-level testing are common, as is "software rot" that gradually appears as systems go through a long series of changes. Although these problems are undoubtably due in part to human error, the counterexamples to the distributivity law suggest that there may be other factors involved as well. At the current state of the art it would take a tedious experimental effort to accurately gauge how important these effects are in actual practice. This should become easier after reliable tools for semantically-based change merging are developed that can adequately treat at least one programming language that is used in large software development and maintenance activities.

It is also natural to ask what practitioners can do to avoid merging problems before reliable change merging tools become available. Our results indicate that in the general case changes cannot be combined reliably without including the context of the change. If accurate behavioral specifications for submodules are available, then these can be used to break some of the dependency chains, thus reducing the size of the part of the context that must be considered when implementing and checking a change [15]. This enables a higher degree of concurrency between the work of designers working on different changes without giving up serializability of the updates. In such an approach each change must be checked with respect to only one context (the one defined by the serialization order, and identified as the primary input of the top level step in [15]) because each change becomes part of the context for the next change in the serialization order.

## 5. Conclusions and Future Work

We have provided a characterization of the semantic properties of an operation for combining changes to software objects, and have shown that the formal model has some of the properties that we would intuitively expect. This characterization is independent of the programming language in which the software objects are described, and can be applied in many contexts. We note that the HPR algorithm [12] is correct with respect to our characterization for the cases in which it does not report any conflicts. One advantage of our formal model is that it provides a natural representation for conflicts in the cases where the given changes cannot be consistently combined. The model can be applied to requirements and specifications in addition to programs, if we accept the view that a specification is a predicate that characterizes the set of all acceptable system behaviors, although the details of this are not explored in the current paper.

In related work [6] we have applied our model of the change combination process to propose a new method for integrating changes to PSDL programs. PSDL is a language for prototyping large, real-time systems, which is based on an enhanced data flow model of computation [14]. This language includes features for expressing concurrency and real-time constraints. A formal semantics of PSDL can be found in [13]. An initial version of a method for combining changes to PSDL programs has been developed [6, 8]. We are also investigating the application of this framework to the development of program transformations that change the semantics of a program in a disciplined way [4]. Such transformations are important in software evolution, and form a complement to the meaning-preserving transformations that are used in implementing executable specification languages and in program optimization.

The results presented in this paper establish a clear correctness criterion for software change merging. Much more work remains to be done before automated change merging can be realized in practice. In particular, the fact that the program merging operation does not distribute over functional composition (see Section 4.4) implies that changes to "independent" modules cannot in general be processed independently. This may provide some objective support for the common belief that software maintenance is difficult to do, and suggests that significant computational capacity may be required to reliably combine changes to large software systems. Section 4.4 determines some of the conditions under which changes to subfunctions can be merged independently.

Since the exact solution to the program merging problem is undecidable, future work should explore the tradeoff between computational efficiency and the size of the subspace for which an algorithm can produce exact results. We would like to have reliable approximation algorithms that produce few spurious warnings and succeed in finding conflict-free merges in most of the cases where they exist. An initial step in this direction has been based on program dependency graphs [12]. A calculus for deriving change merges with fewer spurious conflicts and the capability to successfully merge speedup transformations with other changes is described in [2]. More work is needed to develop representations of programs that can support accurate program merge algorithms with wide coverage. The crux of the problem is to recognize equivalent program fragments that are not identical. Normal forms and transformation technology [9] are likely to be important for this aspect of the problem.

## Acknowledgement

The author would like to thank Mike Mislove for kindly communicating that non-distributivity of the Egli-Milner power domain is part of the folklore of domain theory and to thank the referees for their perceptive comments, which led to substantial improvements to the paper.

## References

1. V. Berzins, "On Merging Software Extensions", *Acta Informatica 23*, Fasc. 6 (Nov. 1986), 607-619.

2. V. Berzins, "Software Merge: Models and Methods", *International Journal on Systems Integration 1*, 2 (Aug. 1991), 121-141.

3. V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.

4. V. Berzins, Luqi and A. Yehudai, "Using Transformations in Specification-Based Prototyping", *IEEE Trans. on Software Eng. 19*, 5 (May 1993), 436-452.

5. V. Berzins, ed., *Proc. ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Software Slicing, Merging, and Integration*, U.S. Naval Postgraduate School, Monterey, CA, Oct. 1993.

6. D. Dampier, "A Model for Merging Different Versions of a PSDL Program", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1990.

7. D. Dampier, Luqi and V. Berzins, "Automated Merging of Software Prototypes", in *Proc. of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, June 16-18, 1993, 604-611.

8. D. Dampier, Luqi and V. Berzins, "Automated Merging of Software Prototypes", *Journal of Systems Integration*, to appear.

9. J. V. Guttag, D. Kapur and D. R. Musser, "Derived Pairs, Overlap Closures, and Rewrite Dominoes: New Tools for Analyzing Term Rewriting Systems", in *Lecture Notes in Computer Science*, Springer-Verlag, 1982, 300-312.

10. P. Halmos, *Lectures on Boolean Algebras*, Van Nostrand, Princeton, NJ, 1963.

11. C. A. R. Hoare, "A Couple of Novelties in the Propositional Calculus", *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik 31*, 2 (1985), 173-178.

12. S. Horowitz, J. Prins and T. Reps, "Integrating Non-Interfering Versions of Programs", *Trans. Prog. Lang and Systems 11*, 3 (July 1989), 345-387.

13. B. Kraemer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Trans. on Software Eng. 19*, 5 (May 1993), 453-477.

14. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng. 14*, 10 (October, 1988), 1409-1423.

15. Luqi, "A Graph Model for Software Evolution", *IEEE Trans. on Software Eng. 16*, 8 (Aug. 1990), 917-927.

16. H. MacNeille, "Partially Ordered Sets", *Transactions of the American Mathematical Society 42* (1937), 416-460.

17. S. Maclane and G. Birkhoff, *Algebra*, Macmillan, New York, 1967.

18. J. McKinsey and A. Tarski, "The Algebra of Topology", *Annals of Mathematics 45*, 1 (Jan. 1944), 141-191.

19. J. McKinsey and A. Tarski, "On Closed Elements in Closure Alggebras", *Annals of Mathematics 47*, 1 (Jan. 1946), 122 - 162.

20. G. Ramalingam and T. Reps, "A Theory of Program Modifications", in *Proc. of the Colloquium on Combining Paradigms for Software Development*, vol. LNCS 494 , Springer-Verlag, Apr. 1991, 137-152.

21. T. Reps, "Algebraic Properties of Program Integration", *Science of Computer Programming 17*, 1-3 (Dec. 1991), 139-215.

22. C. Rich and R. Waters, *The Programmer's Apprentice*, Addison Wesley, Reading, MA, 1990.

23. W. Roscoe, *Lecture Notes on Domain Theory*, Programming Research Group, Oxford University, 1992.

24. L. Sterling and A. Lakhotia, "Composing Prolog Meta-Interpreters", in *Logic Programming: Proc. of the Fifth Int. Conf. and Symposium*, MIT Press, 1988, 386- 403.

25. J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

26. W. Yang, S. Horowitz and T. Reps, "A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations", in *Proc. 4th ACM Software Eng. Notes Symposium on Software Development Environments*, Irvine, CA, Dec. 1990, 133-143.

## Distribution List

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 2

Library, Code 52
Naval Postgraduate School
Monterey, CA 93943 2

Research Office
Code 08
Naval Postgraduate School
Monterey, CA 93943 1

Dr. Luqi, Code CS Lq
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943-5118 10

Prof. Valdis Berzins, Code CSBe
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943-5118 30